

This is the last unit of the tutorial, and while many of the details about productions and modules have been covered in the tutorial, there are some things which were not. You can find more details on productions and other module actions that were not used in the tutorial in the reference manual.

Productions

Productions are the representation of procedural knowledge in ACT-R, and are an if-then rule consisting of a condition and an action. If the condition is true then it will perform the action. The condition is a set of patterns to compare to the model's buffers, and the actions are changes that are made to the buffers. The basic cycle of a model's operation is to find a production with a true condition, perform its action, and repeat that process until there is nothing else it can do.

Syntax and notation details

General details and definitions:

- Any white space characters (spaces, tabs, returns, and newlines) can be used to separate items in a production. The amount and type doesn't matter.
- A name is a continuous sequence of characters which are not all numbers and does not include any of these characters: , () ` ' " # :
- A name is not case sensitive. Goal, goal, and GOAL are all the same name in a production.
- A string is any sequence of characters between a pair of double quotes: " ... "
- Names and strings are not the same in a production. The name a is not the same as the string "a".

When describing the syntax of items the following conventions will be used:

- **bold** items must be specified exactly as indicated
- *italic* items are optional
- Angle brackets < > around items forms a group of items that occur together
- Curly brackets { } indicate a set of possible values
- Anything followed by a * means that there may be any number of those items (including zero)
- Anything followed by a + means that there must be at least one of those items, but can be more

Syntax

production → (**p** pname *documentation* condition ==> action)

pname → a name which must be unique for each production

documentation → a string

condition → {pattern, user-tool }*

pattern → =buffer> <**isa** type> <modifier slot-name value>*

pattern → ?buffer> <modifier query-test query-value>+

action → {change, user-tool }*

change → =buffer> <**isa** type> <slot-name value>*

change → *buffer> <**isa** type> <slot-name value>*

change → +buffer> <**isa** type> <param value>* <modifier slot-name value>*

change → +buffer> cname

change → -buffer>

change → @buffer>

user-tool → **!output!** (value*) only as an action

user-tool → **!eval!** (command value*)

user-tool → **!bind!** variable (command value*)

user-tool → **!safe-eval!** (command value*)

user-tool → **!safe-bind!** variable (command value*)

buffer → { **goal**, **retrieval**, **imaginal**, **manual**, **visual-location**, **visual**,
aural, **aural-location**, **vocal**, **imaginal-action** }

type → a name of a defined chunk-type

modifier → { -, <, >, <=, >= }

slot-name → { name, variable }

value → { variable, name, string, number, **nil** }

variable → =name

param → :name

query-test → {**buffer**, **state**}

query-test → {**buffer**, **state**, **preparation**} if buffer is **manual** or **vocal**

query-value → {**full**, **failure**, **empty**, **requested**, **unrequested**} if query-test
is **buffer**

query-value → {**free**, **busy**} if query-test is **state** or **preparation**

command → a string naming a valid ACT-R command

cname → variable with the value that is the name of a chunk

cname → the name of a chunk

Patterns

=buffer>

A pattern that starts with the = character is a test that there is a chunk in a buffer with a specific set of slots and values. However if the procedural partial matching mechanisms is enabled then the pattern may match to values that are similar to those specified instead of having to match everything exactly. It is true if there is a chunk in the named buffer and all of the slots and values provided in the pattern match those in that chunk. Variables can be used to test that multiple slots have the same value. The value **nil** can be used to test that a chunk does not have the indicated slot. The - modifier can be used to test that a slot does not have the value indicated. If a chunk-type is provided that has default values for some slots, then those slots and values will be part of the pattern unless the slot is explicitly specified in the pattern. If the pattern matches then the =buffer variable will be set to the name of the chunk in the buffer.

Examples:

```
=goal> isa type1 slot1 =s - slot2 b =s "s"
```

This is a test of the chunk in the goal buffer. It indicates that the slots being used are from a chunk-type named type1. Slot1 of the chunk must a value which sets the =s variable, slot2 of the chunk must not have the value b because of the - modifier, and the slot named by the value of =s must be the string "s".

```
=retrieval> slot3 nil slot4 =x slot5 =x
```

This is a test of the chunk in the retrieval buffer. It does not specify a chunk-type for the slots. There must not be a value in the slot named slot3 of the chunk because the value tested is **nil**, and the slot4 and slot5 slots must have the same value because they are tested with the same variable.

```
=visual-location> > screen-x 100 <= screen-x 200
```

This is a test of the chunk in the visual-location buffer. It does not specify a chunk-type for the slots. There must be a number in the slot named screen-x of the chunk which has a value greater than 100 and also less-than or equal to 200.

?buffer>

A pattern that starts with the ? character is a test of the status of the buffer itself and/or the buffer's module, which is called a query. The - modifier can be used to test that indicated status is not true.

Examples:

```
?retrieval> state free buffer full
```

```
?goal> - buffer empty
```

The first is a query of the retrieval buffer and its module (the declarative memory module). It is true if the declarative module indicates that its state is free (it is not performing an action) and the retrieval buffer currently contains a chunk. The second is a query of the goal buffer and is true if the buffer is not empty because of the - modifier.

Variables

All variable matched in the condition will have that same value in the production's action.

Changes

=buffer>

A change that starts with the = character is a modification of the chunk in the buffer. The specified slots will be set to the values provided, except if the value is **nil**. If a chunk-type is provided that has default values for some slots, then those slots will be set to those values unless the slot is explicitly specified. The value **nil** will remove the slot from the chunk.

Example:

```
=goal> isa type1 slot1 c slot2 =x slot4 nil =x 3
```

The slots being modified are declared to be in the chunk-type named type1. It will change the value of the slot1 slot of the chunk in the goal buffer to c, the slot2 slot will have the value of the =x variable, the slot4 slot will be removed from the chunk, and the slot named by the value of the =x variable will be set to 3.

-buffer>

A change that starts with the - character will remove the chunk from the buffer, and it will then be stored in the model's declarative memory.

Example:

```
-goal>
```

This will remove the chunk from the goal buffer and it will be stored in declarative memory.

@buffer>

A change that starts with the @ character will remove the chunk from the buffer and unlike the – action will prevent it from being stored in the model's declarative memory. This is not generally recommended, but there are situations where it can be useful.

Example:

```
@goal>
```

This will remove the chunk from the goal buffer without storing it in declarative memory.

+buffer>

A change that starts with the + character is a request that the buffer's module perform some action. This will also clear the buffer if it currently has a chunk in it. Each module handles requests differently and can have different requirements for how the request is specified. If a chunk-type is provided that has

default values for some slots, then those slots and values will be part of the request unless the slot is explicitly specified.

All **goal** buffer requests will create a new chunk with the slots and values specified (the modifier is not allowed for any slots in the request) and it will place that chunk into the goal buffer.

All **retrieval** buffer requests will search the model's declarative memory to find a chunk which matches the pattern specified (the same way a buffer test in the production's condition matches chunks except that when the partial-matching mechanism is enabled chunks which don't perfectly match the pattern may also be retrieved). The request parameter **:recently-retrieved** can be specified with the value **nil** indicating that only items not marked as having been retrieved recently will be considered, or **t** indicating only the items marked as having been retrieved recently are considered. If more than one chunk matches the pattern then the chunk with the highest activation value is selected. If a chunk is selected which has an activation value at or above the retrieval threshold parameter value then it will be put into the retrieval buffer after a time determined by the activation. If no matching chunk is found or the matching chunk has an activation value below the threshold then the retrieval buffer will be marked with a failure after a time that depends on the value of the retrieval threshold.

All **imaginal** buffer requests will create a new chunk with the slots and values specified like the goal buffer. However, there is a 0.2 second cost to the imaginal module's action (by default) during which time the imaginal buffer's state busy query will be true and it will not be able to accept another request.

All **imaginal-action** buffer requests can be used to call external code (much like a !eval!) which is intended to be used for user extensions to the imaginal module. Those actions should modify or replace the chunk that is in the imaginal buffer. The imaginal-action buffer is not intended to be used for holding chunks.

All **visual-location** buffer requests cause the vision module to search the visicon to find a feature that matches the description specified in the slots and values provided. If the **:attended** request-parameter is specified that will restrict which items are searched based on the attended status indicated. If the **:nearest** request-parameter is specified with a value that represents a visual location or the value **current** which means the location currently being attended, then if multiple items are found in the search the item nearest to that specified location will be put in the buffer. If a feature matching the request is found then a chunk is created to represent that feature and placed into the visual-location buffer. If no matching feature is found then the buffer will be marked with a failure.

Requests to the **visual** buffer require specifying a slot named **cmd** whose value will indicate the action for the vision module to perform. The value **move-attention** will cause the vision module to shift its attention to the location specified in the **screen-pos** slot of the request (which is typically the value **=visual-location** from matching the pattern in the visual-location buffer in the condition). If there is a visual object at the location attention is shifted to then a chunk representing that object is placed in the visual buffer. If no visual object is found then the visual buffer will be marked with a failure. That attention shift takes 0.085 seconds (by default) during which time the visual buffer's state busy query will be true and it will not be able to accept another request. A **cmd** slot value of **clear** will make the vision module stop attending to an item which it is currently attending and will make the visual buffer's state busy query true for 0.05 seconds.

Requests to the **manual** buffer require specifying a slot named **cmd** whose value will indicate the action for the motor module to perform. The value **press-key** will cause the motor module to press the key specified in the **key** slot of the request on the keyboard device (if it is installed). That action will take time (which depends on which key and the prior motor action) during which the buffer's state busy query will be true and it may not be able to accept another request. If the manual buffer's preparation free query is true then it will be able to accept a new request even if the state busy query is also true.

All **aural-location** buffer requests cause the audio module to search the audicon to find a sound event that matches the description specified in the slots and values provided. If the **:attended** request-parameter is specified that will restrict which items are searched based on the attended status indicated. If an event matching the request is found then a chunk is created to represent that event and placed into the aural-location buffer. If no matching feature is found then the buffer will be marked with a failure.

Requests to the **aural** buffer specifying a slot named **event** with a value that is a sound event will cause the audio module to shift its attention to that event. If there is still a sound available for that event then the model's aural attention is shifted to that sound and a chunk representing it is placed in the aural buffer. If no corresponding sound is found then the aural buffer will be marked with a failure. That attention shift takes time (which varies depending on the type of sound) during which the aural buffer's state busy query will be true and it will not be able to accept another request.

Requests to the **vocal** buffer require specifying a slot named **cmd** whose value will indicate the action for the speech module to perform. The value **speak** will cause the speech module to speak the string specified in the **string** slot of the request which will be detected by the microphone device (if it is installed). That action will take time (depending on the content of the string provided and the prior speech action) during which the buffer's state busy query will be true and it may not be able to accept another request. A **cmd** slot value of **subvocalize** will cause the speech module to subvocalize the string specified in the **string** slot of the request. A subvocalize action is not detectable by the microphone device and will take time (which depends on the content of the string provided and the prior speech action) during which the buffer's state busy query will be true and it may not be able to accept another request. If the vocal buffer's preparation free query is true then it will be able to accept a new request even if the state busy query is also true.

Examples:

```
+goal> isa type2 slot1 blue slot2 =z
```

This request will clear the chunk from the goal buffer if there is one. Then the goal module will create a chunk with two slots, slot1 having the value blue and slot2 having the value that the =z variable was matched to in the production's condition. That chunk will then be placed into the goal buffer.

```
+retrieval> slot1 green slot3 =x - slot4 3
```

This request will clear the chunk from the retrieval buffer if there is one. Then the declarative module will search the chunks it has stored to find one which has a slot named slot1 with the value green, a slot named slot3 having the value that the =x variable was matched to in the production's condition, and a

slot4 value that is not the number 3. That chunk will then be placed into the retrieval buffer. If no chunk is found matching that pattern then the retrieval buffer will indicate a failure.

```
+visual-location> :attended nil color blue
```

This request will clear the chunk from the visual-location buffer if there is one. Then the vision module will search for a feature in the visicon which the model has not yet attended (because of the :attended request parameter specified as **nil**) which is the color blue. If there is such a feature the vision module will create a chunk to represent it and place it into the visual-location buffer. If no feature was found the visual-location buffer will indicate a failure.

```
+vocal> cmd speak string "word"
```

This request will clear the chunk from the vocal buffer if there is one. Then the speech module will speak the indicated string "word".

```
+retrieval> =val
```

This request will clear the chunk from the retrieval buffer if there is one. Then a request consisting of all the slots and values from the chunk name that is matched to the =val variable will be sent to the declarative module. This will cause the module to try to find a chunk in memory which looks like that chunk, but if partial matching is enable even if that specific chunk is in declarative memory, it may not be the one that is retrieved and placed it into the retrieval buffer.

```
*buffer>
```

A change that starts with the * character is a request that a module perform a modification of the chunk in the buffer. If a chunk-type is provided that has default values for some slots, then those slots and values will be part of the request unless the slot is explicitly specified. Only the **goal** and **imaginal** modules support this action.

Example:

```
*imaginal> isa type1 slot1 c slot2 =x slot4 nil
```

The slots being modified are declared to be in the chunk-type named type1. The imaginal module will change the value of the slot1 slot of the chunk in the goal buffer to c, the slot2 slot will have the value of the =x variable, and the slot4 slot will be removed from the chunk. This will take 0.2 seconds to finish (by default) during which time the imaginal buffer's state busy query will be true and it will not be able to accept another request.

User-tools

There are special operations that can be used in the condition and action of a production which are not part of the ACT-R theory, but are software tools that can be helpful when creating models.

!output!

The **!output!** operation can be used in the action of a production to display information in the model trace. When that production is selected and performs its action those items will be displayed on a line of the trace.

Example:

!output! (the variable has the value =x)

If the =x variable matched to the value 3 in the condition of the production then the trace will show this when the production performs its action:

```
the variable has the value 3
```

!eval! and !safe-eval!

The **!eval!** operation can be used in the condition and action of a production to call an ACT-R command. The provided values will be passed to that command. If it is in the condition then the command must return a true value for the production to be selected. Anything which is not **nil** in Lisp or the equivalent in another language (**False** or **None** in Python for example) will be considered true. If production compilation is being used **!eval!** will prevent composition of a production that uses it. However, if the command is “safe” (does not do anything that would change the operation of the model when composed into a new production) you can use **!safe-eval!** instead to allow the composition.

Example:

!eval! ("test-command" 1 =var)

This operator in a production will call the command named test-command passing it two values, the number 1 and the value to which the =var variable matched in the production.

!bind! and !safe-bind!

The **!bind!** Operator can be used in a production to call a command available in ACT-R just like **!eval!**. The difference is that **!bind!** also records the return value of the evaluation in a production variable. If the return value is **nil** then the variable will be set to the value **t** instead since a variable must have a value. If production compilation is being used **!bind!** will prevent composition of a production that uses it. However, if the command is “safe” (it does not do anything that would change the operation of the model

when composed into a new production and removing the !bind! and replacing the bound variable with its instantiation during composition like a retrieval request does for its variables will not affect the model's operation) you can use !safe-bind! instead to allow the composition.

Example:

```
!bind! =result ("test-command" 1 =var)
```

This action in a production will call the command named test-command passing it two values, the number 1 and the value to which the =var variable matched in the production. The =result variable will have the value that was returned from calling the command.